To use the K-Nearest Neighbor classifier, a module was written in python called knn which contains the methods used for preparing the data, finding neighbors, polling neighbors, creating the dev, train, and test data splits, along with the methods for Euclidean and Manhattan distances. This module is loaded into the main.py script which handles getting the data from Sklearn, calling the methods within the knn module, and then creating plots using pyplot from Matplotlib.

Using this script, the optimal K value was determined over multiple iterations from 1 to 80 at increments of 3, with the accuracy score for the given K value saved for each distance equation (Manhattan and Euclidean) and plotted to a chart comparing them. This process was repeated 5 times to determine that the accuracy score of the KNN decreased in all instances after K=15. From there, the process was repeated but as 1 to 15 at an increment of 1. Using that, the optimal K value was found to be K=1 across multiple. Initially this was thought to be an error, potentially with the data shuffling, but after multiple iterations where latter K values were different accuracies, it was determined to not be from randomization. Rationale for accepting K=1 is that the separation between labels is significantly large enough that few pieces of data fall into an edge case, and by polling larger swaths of neighbors (increasing K) the chance of overlap between these distinct labels increase. Also observed was that in lower K values, Euclidean distance always performed with higher accuracy than Manhattan distance but in each instance, there was a point where K became substantially large, and the accuracy of Manhattan distance matched or was greater than the accuracy score of the Euclidian distance for those large K values.

Chart, line chart

Description automatically generated

Figure - Dev Set Accuracy Scores Run 1

Chart, line chart

Description automatically generated

Figure - Dev Set Accuracy Scores Run 2

Chart, line chart

Description automatically generated

Figure - Dev Set Accuracy Scores Run 3

Chart, line chart

Description automatically generated

Figure - Dev Set Accuracy Scores Run 4

Chart, line chart

Description automatically generated

Figure - Dev Set Accuracy Scores Run 5

Chart, line chart

Description automatically generated

Figure - Dev Set Accuracy Scores Refined Run 1

Chart, line chart

Description automatically generated

Figure - Figure 6 - Dev Set Accuracy Scores Refined Run 2

Chart, line chart

Description automatically generated

Figure - Figure 6 - Dev Set Accuracy Scores Refined Run 3

Chart, line chart

Description automatically generated

Figure - Figure 6 - Dev Set Accuracy Scores Refined Run 4

Chart, line chart

Description automatically generated

Figure - Figure 6 - Dev Set Accuracy Scores Refined Run 5

Following this refinement, the accuracy score of K = 1 did vary but averaged at 95% accuracy.

The output is below, and the piece of code used to generate the output is defined as test\_accuracy() in main.py.

Text

Description automatically generated

Figure - Test Set Accuracy Scores

Using the K value = 1, 10 random points were selected from the test set, and accuracy was 90% with only 1 incorrect prediction. The code snipped used is defined as random\_10\_tests() in main.py

Text

Description automatically generated

Figure - 10 Random Test Points Accuracy

knn.py:

from sklearn.metrics import accuracy\_score

import numpy as np

'''

Modified from lecture, converted into a reusable function

'''

def train\_dev\_test\_split(data, p\_train, p\_dev):

total\_number = len(data)

# convert percentage to decimal

p\_train = p\_train/100

# convert percentage to decimal, append percentage from train

p\_dev = p\_dev/100 + p\_train

# train off first x percentage

train = data[:int(total\_number \* p\_train)]

# dev off the next x percent

dev = data[int(total\_number\*p\_train):int(total\_number\*p\_dev)]

# test off remaining percent

test = data[int(total\_number\*p\_dev):]

# return all portions of data

return train, dev, test

'''

From Lecture, I do not claim to have written this piece of code

'''

def get\_features\_and\_labels(data):

features = data[:, :-1]

labels = data[:, -1]

return features, labels

'''

From Lecture, I do not claim to have written this piece of code

'''

def data\_leaking\_check(data1, data2):

data\_leaking = False

for d1 in data1:

for d2 in data2:

if(np.array\_equal(d1, d2)):

data\_leaking = True

print("Find same sample: ")

print(d1)

if(not data\_leaking):

print("No Data Leaking!")

'''

From Lecture, I do not claim to have written this piece of code

'''

def euclidean\_distance(row1, row2):

distance = 0.0

for i in range(len(row1)-1):

distance += (row1[i] - row2[i]) \*\* 2

return np.sqrt(distance)

'''

Implementation of the manhattan distance equation

'''

def manhattan\_distance(row1, row2):

distance = 0.0

for i in range(len(row1)-1):

distance += abs(row1[i] - row2[i])

return distance

'''

Modified from lecture to allow distance metric as a parameter

'''

def get\_neighbors(train\_x, train\_y, test\_row, num\_neighbors, dist\_func):

distances = []

# get all distances

for index in range(len(train\_x)):

train\_row = train\_x[index]

train\_label = train\_y[index]

dist = dist\_func(train\_row, test\_row)

distances.append((train\_row, train\_label, dist))

# sort the distance list by distance

distances.sort(key=lambda i: i[2])

# get the k nearest neightbors and return

output\_neighbors = []

output\_labels = []

output\_distances = []

for index in range(num\_neighbors):

output\_neighbors.append(distances[index][0])

output\_labels.append(distances[index][1])

output\_distances.append(distances[index][2])

return output\_neighbors, output\_labels, output\_distances

'''

From Lecture, I do not claim to have written this piece of code

'''

def prediction\_classification(train\_x, train\_y, test\_row, num\_neighbors, distance\_function):

output\_neighbors, output\_labels, output\_distances = get\_neighbors(train\_x, train\_y, test\_row, num\_neighbors, distance\_function)

label\_cnt = np.bincount(output\_labels)

prediction = np.argmax(label\_cnt)

return prediction

main.py:

import knn

from sklearn.datasets import load\_digits

from random import randint

import matplotlib.pyplot as plt

import numpy as np

# Find Highest Performing K for both distance equations

def get\_metrics(train\_x, train\_y, input\_x, input\_y, input\_name, k\_max, k\_inc, metric\_function, metric\_name, iteration):

# generate list of K from 1 to max under given increment

k\_list = list(range(1, k\_max, k\_inc))

# initialize empty performance arrays

performance\_euclidean = []

performance\_manhattan = []

# iterate over K values

for k in k\_list:

# Notice of Progress

print('Testing K value: %s Metric: %s'%(k, metric\_name))

# Initialize empty prediction arrays

pred\_labels\_e = []

pred\_labels\_m = []

# iterate over data in input set

for data in input\_x:

# get predictions for distance functions

pred\_e = knn.prediction\_classification(train\_x, train\_y, data, k, knn.euclidean\_distance)

pred\_m = knn.prediction\_classification(train\_x, train\_y, data, k, knn.manhattan\_distance)

# append to prediction arrays

pred\_labels\_e.append(pred\_e)

pred\_labels\_m.append(pred\_m)

# evaluate metrics

met\_e = metric\_function(input\_y, pred\_labels\_e)

met\_m = metric\_function(input\_y, pred\_labels\_m)

print(met\_e, met\_m)

#append to metrics array

performance\_euclidean.append(met\_e)

performance\_manhattan.append(met\_m)

# Initialize figure

plt.figure(figsize=(20,6))

# Add plots for distance functions

plt.plot(k\_list, performance\_euclidean, marker="o", label="Euclidean Distance")

plt.plot(k\_list, performance\_manhattan, marker="o", label="Manhattan Distance")

# Add Legend

plt.legend()

plt.xlabel("K Values")

plt.ylabel("Accuracy")

plt.title("Performace on %s Set"%(input\_name))

plt.savefig("%s\_%s\_%s.png"%(input\_name, metric\_name, iteration))

def test\_accuracy():

# Prediction array

predictions = []

# Iterate over the data

for data in test\_x:

pred = knn.prediction\_classification(train\_x, train\_y, data, 1, knn.euclidean\_distance)

predictions.append(pred)

# Generate the Score

score = knn.accuracy\_score(test\_y, predictions)

# Output the result

print("Test Set Accuracy Score: %s"%(score))

def random\_10\_tests():

# 10 Values

for \_ in range (10):

# Generate random index

index = randint(0, len(test\_x) -1)

# Get test row

x = test\_x[index]

# Get real label value

y = test\_y[index]

# Get prediction

pred = knn.prediction\_classification(train\_x, train\_y, x, 1, knn.euclidean\_distance)

# Visualize Prediction vs Actual

print("Predicted Label: %s, Actual Label: %s"%(pred, y))

# Load the digits

digits = load\_digits()

# Grab Data and Labels

data = digits.data

labels = digits.target

# Adjust dimensions

labels = np.expand\_dims(labels, 1)

# Append

data = np.append(data, labels, 1)

# Randomize the data

np.random.shuffle(data)

# create dev, train, test sets

dev, train, test = knn.train\_dev\_test\_split(data, 70, 15)

# check for data leaks

knn.data\_leaking\_check(dev, train)

knn.data\_leaking\_check(dev, test)

knn.data\_leaking\_check(train, test)

# split into x, y

dev\_x, dev\_y = knn.get\_features\_and\_labels(dev)

train\_x, train\_y = knn.get\_features\_and\_labels(train)

test\_x, test\_y = knn.get\_features\_and\_labels(test)

# Dev Metrics

#get\_metrics(train\_x, train\_y, dev\_x, dev\_y, "Dev", 15, 1, knn.accuracy\_score, "Accuracy", 10)

# Test Set Accuracy

test\_accuracy()